TP Classe Entier

1. *Préparation*

A l'aide de QtCreator ou Visual C++, coder la classe *TabEntiers* vue en TD :

```
#include <iostream>
#include <stdlib.h>

using namespace std;

const int NbMax = 100;

class TabEntiers
{
    private :
        int taille;
        int nbElem;
        int *table;
        void alloueTable(int Dimension)
        {
            table = new int[Dimension];
            if (table == NULL)
            {
                cerr << "TableEntiers : allocation impossible\n";
                exit(1);
            }
            taille = Dimension;
        }
    public :
        TabEntiers (int Dim)
        {
            alloueTable(Dim);
            nbElem = 0;
        }
};
```

Remarques :

- Vous pouvez remarquer que le code des méthodes *alloueTable()* et *TabEntiers()* est dans la classe. C'est une possibilité d'écriture du C++ : ce sont des fonctions dites « inline ».
  Cela pose un problème : le code est visible puisqu'on fournit généralement les fichiers .h. Il est préférable de séparer le code dans un .cpp qui sera compilé donc non éditable.
- On crée une méthode privée *alloueTable()* qui s'occupe d'effectuer la réservation mémoire. Ainsi, si on a plusieurs constructeurs, on n'effectuera que l'appel à cette méthode.

2. Modification

Faites en sorte que le code des méthodes soit séparé de la classe (donc dans un .cpp).

Réaliser une fonction main de test comme ceci :

```
#include <iostream>
#include "TabEntiers.h"

int main(int argc, char *argv[])
{
    TabEntiers Tab1(50), Tab2(100), Tab3;
    return 0;
}
```

Si on compile « à la main » et on constate qu'une erreur est produite :

```
E:\lycee\SN2IR\TP\ClasseEntier>g++ -o test.exe test.cpp TabEntiers.cpp
test.cpp: In function 'int main(int, char**)':
test.cpp:6:34: error: no matching function for call to 'TabEntiers::TabEntiers()
'
  TabEntiers Tab1(50), Tab2(100), Tab3;
                                  ^
test.cpp:6:34: note: candidates are:
In file included from test.cpp:2:0:
TabEntiers.h:16:3: note: TabEntiers::TabEntiers(int)
   TabEntiers (int);
   ^
TabEntiers.h:16:3: note:   candidate expects 1 argument, 0 provided
TabEntiers.h:8:7: note: TabEntiers::TabEntiers(const TabEntiers&)
 class TabEntiers
       ^
TabEntiers.h:8:7: note:   candidate expects 1 argument, 0 provided
```

C'est l'illustration de ce qu'on a vu en cours : le compilateur ne trouve pas de constructeur pour construire l'objet *Tab3* .Le constructeur par défaut n'est plus fourni par le C++ à partir du moment où on a créé un constructeur avec un paramètre.

Ajouter une valeur par défaut au paramètre *Dim* . Attention : contrairement à ce qu'on a vu en TD, la valeur par défaut n'est pas précisée dans la signature de la méthode mais dans la signature du prototype de la méthode.

3. Discussion autour des paramètres par défaut

On désire ajouter un constructeur prenant en paramètre non seulement la taille du tableau mais aussi une valeur qui permettra d'initialiser le tableau avec celle-ci.

On voudrait laisser la possibilité d'avoir une valeur par défaut. Se pose alors à nous plusieurs problèmes.

Voici différentes signatures de prototypes de constructeurs :

Essai 1

```
TabEntiers (int Dim = 100);
TabEntiers (int Dim, int Val = 0);
```

Cela signifie qu'on voudrait créer un objet en précisant la taille et/ou la taille et la valeur et/ou aucun paramètre. Mais ça ne fonctionne pas. Pourquoi ?

Regardons les signatures possibles :

TabEntiers (int Dim = 100) => TabEntiers(int) et TabEntiers()

TabEntiers(int Dim, int Val = 0) => TabEntiers(int, int) et TabEntiers(int)

On vient bien que le compilateur a deux signatures semblables (en rouge) pour les deux constructeurs donc il ne sait pas lequel exécuter. C'est ce qui se passe si on cherche à instancier un TabEntiers en faisant :

TabEntiers Tab(50) ;

Essai 2

```
TabEntiers (int Dim = 100);
TabEntiers (int Dim, int Val);
```

TabEntiers (int Dim = 100) => TabEntiers(int) et TabEntiers()

TabEntiers(int Dim, int Val) => TabEntiers(int, int)

Là pas d'ambiguïté : Aucune des trois signatures n'est semblable aux autres.

Essai 3

```
TabEntiers (int Dim = 100);
TabEntiers (int Dim = 100, int Val);
```

TabEntiers (int Dim = 100) => TabEntiers(int) et TabEntiers()

TabEntiers(int Dim = 100, int Val) => TabEntiers(int, int) et TabEntiers(int)

Donc ambiguïté entre les deux signatures rouges.

4. Ajout de méthodes

4.1 Constructeur TabEntiers(int Dim, int Val)

Le constructeur TabEntiers crée un tableau de la taille Dim et initialise tous les éléments du tableau avec la valeur Val.

4.2 Méthode void afficheTab(..)

Elle affiche le contenu du tableau.

4.3 Méthode getMaxTab(..)

Elle renvoie la valeur max du tableau de l'objet.

4.4 Méthode remplitAleaTab()

Elle effectue un remplissage du tableau avec des valeurs tirées aléatoirement.

4.5 Constructeur TabEntiers(TabEntiers &)

Le constructeur crée un objet TabEntiers à partir d'un autre objet du même type passé par référence.

Remarque : Vous serez amenés à ajouter d'autres méthodes, par exemple pour connaître la taille du tableau ou connaître la valeur d'un élément du tableau.

4.6 Méthode resizeTab()

La méthode *ResizeTab()* permet d'augmenter ou de diminuer la taille du tableau tout en conservant les éléments déjà enregistrés.

Augmentation de taille : on ajoute 3 éléments

2	5	1	6	9

2	5	1	6	9	X	X	X

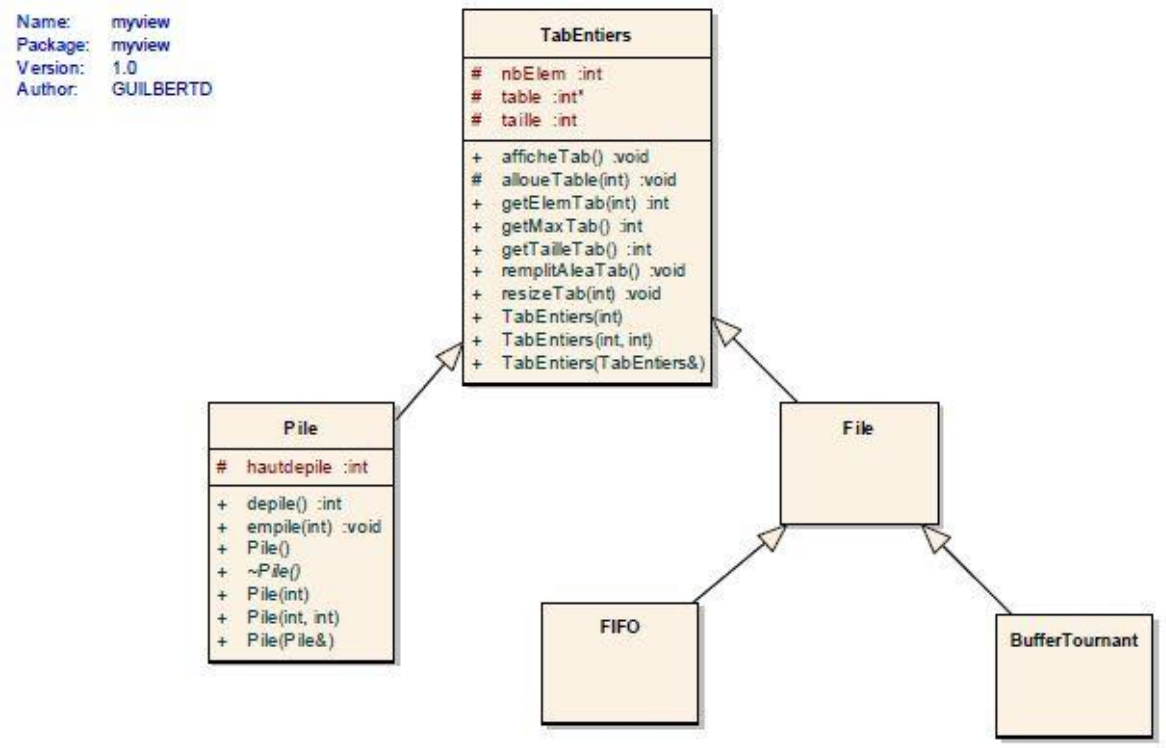
Diminution de taille : on retire 2 éléments

2	5	1	6	9

2	5	1

Les éléments 6 et 9 sont perdus !

Classe Pile (LIFO)



La classe Pile hérite de la classe TabEntiers. Elle modélise le concept de pile qui consiste à pouvoir empiler des valeurs et dépiler celles-ci selon le principe « dernier entré premier sorti » (Last In First Out = LIFO)

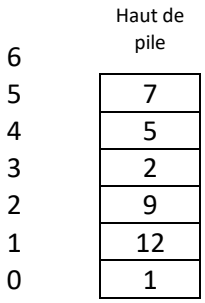
Grâce à cet héritage, la classe Pile dispose déjà de tous les mécanismes de création de tableau, de redimensionnement, etc.. que nous avons développés dans la classe TabEntiers.

Elle ajoute simplement les mécanismes propres à la pile, à savoir : empiler et depiler.

On peut donc, dans une première approche considérer la classe Pile comme :

```
class Pile : public TabEntiers
{
    int hautdepile; //position au dessus du dernier element

public:
    Pile();
    virtual ~Pile();
    int depile();
    void empile(int);
};
```



Haut de pile pointe sur l'indice 6.

1. *Méthode empile(int)*

Elle permet la sauvegarde d'un nouvel élément en haut de la pile. Si le tableau n'est pas assez grand, elle le redimensionne.

2. *Méthode int depile()*

Elle renvoie la valeur du dernier élément en haut de pile. Le tableau n'est pas redimensionné.

3. *Constructeurs de Pile*

Il faut redéfinir les constructeurs de TabEntiers pour Pile. En effet, le constructeur de la classe fille est toujours appelé en second après l'appel du constructeur de la classe mère. Il faut donc bien un constructeur de la classe fille présent dans la classe. Un exemple :

```
Pile::Pile (int Dim) : TabEntiers(Dim)  //Constructeur avec dimension par défaut
{
    hautdepile = 0;
}
```